University of Pittsburgh

Vectorization

Week 3

MEMS 1140—Introduction to Programming in Mechanical Engineering

Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❏ What vectorization is;
- ❏ Why vectorization could be useful in engineering;
- ❏ Vectorized array operations in MATLAB;
- ❏ Write vectorized code for the vector dot product.

Table of Contents (ToC)

1 – Real-life Vectorization

You and some friends plan a cookout grilling hot dogs.

You could grill one at a time, but then some people receive food sooner than others.

Putting an entire pack of hot dogs on the grill simultaneously ensures that more people receive food together.

This is a ***vectorized*** cooking operation.

1 – Definition of Vectorization

The processor (you) has accessed the entire “array” (pack) and applied the function `cook` to the *array as a whole*.

Definition

Vectorized code is characterized by one instruction (one line of code) that operates on multiple data points together.

By utilizing vectorized operations, it is possible to write faster, more reliable code while also making it more readable.

1 – Disclaimer

This lecture blurs the lines a little bit between vectorization and parallelization.

Later lectures will introduce parallelization, but suffice it to say that they *are* different.

A more thorough comparison is provided for the interested reader.

2 – Introduction to Loops

To understand the benefits of vectorization, it is necessary to first examine the alternative: loops.

Loops will be covered in more detail later in this course, but it is useful to define them now for comparison:

Definition

The program executes one or more operations repeatedly until a termination condition is met.

2 – Looping Hot Dogs

In our example grilling hot dogs, a loop would be used to cook each one individually.

It might look as follows:

```
hotdogs(8) = HotDog;                   % array of 8 hot dogs
.
.
.
```

2 – Looping Hot Dogs

In our example grilling hot dogs, a loop would be used to cook each one individually.

It might look as follows:

```
hotdogs(8) = HotDog;                         % array of 8 hot dogs
for
  .
end
```

2 – Looping Hot Dogs

In our example grilling hot dogs, a loop would be used to cook each one individually.

It might look as follows:

```
hotdogs(8) = HotDog;                    % array of 8 hot dogs
for hotdog_index = 1 : 8
  .
end
```

2 – Looping Hot Dogs

In our example grilling hot dogs, a loop would be used to cook each one individually.

It might look as follows:

```
hotdogs(8) = HotDog;                          % array of 8 hot dogs
for hotdog_index = 1 : 8
  cook(hotdogs(hotdog_index));                % cook each one
end
```

This code executes `cook` on each hot dog one at a time.

2 – How to Read a Loop

```
hotdogs(8) = HotDog;                          % array of 8 hot dogs
for hotdog_index = 1 : 8
  cook(hotdogs(hotdog_index));                % cook one by one
end
```

In plain english, this code reads as follows:

For each hot dog in the pack, numbered 1 to 8, cook each one before beginning to cook the next.

2 – Vectorization for Hot Dogs

In comparison, vectorized code may look something like this:

```
hotdogs(8) = HotDog;                    % array of 8 hot dogs
.
```

2 – Vectorization for Hot Dogs

In comparison, vectorized code may look something like this:

```
hotdogs(8) = HotDog;                      % array of 8 hot dogs
cook(hotdogs);                            % cook all hot dogs
```

For this (fictional) example, the vectorized cooking operation is only one line ... and it is likely faster.

Vectorized code is also often easier to read *because* it is shorter and more closely matches plain language.

2 – Prepare an Experiment

Let’s conduct an experiment to see whether vectorization is *really* that much faster than looping.

Experiment

Multiply 1 million random numbers by 2.

The following function generates an **`Nx1`** array of random values given a *length* (**`num_values`**) as its input:

```
random_number_array = @(num_values)
```

2 – Prepare an Experiment

Let’s conduct an experiment to see whether vectorization is *really* that much faster than looping.

Experiment

Multiply 1 million random numbers by 2.

The following function generates an **`Nx1`** array of random values given a *length* (**`num_values`**) as its input:

```
random_number_array = @(num_values) rand(num_values, 1);
```

2 – Defining a Loop

Multiplying each element of an array by 2 using a loop is as follows:

```
initial_values = random_number_array(1e6);
.
.
.
.
```

2 – Defining a Loop

Multiplying each element of an array by 2 using a loop is as follows:

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) ); % initialize output array
.
.
.
```

2 – Defining a Loop

Multiplying each element of an array by 2 using a loop is as follows:

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
for i = 1 : length(initial_values)              % outline the for loop
  .
end
```

2 – Defining a Loop

Multiplying each element of an array by 2 using a loop is as follows:

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
for i = 1 : length(initial_values)
  output_values(i) = initial_values(i) * 2;     % loop multiplication
end
```

Note: The array `output_values` is initialized ***before*** the loop. It is possible, but much slower, to skip this step.

2 – Vectorized Version

The vectorized version of this operation is significantly simpler:

```
initial_values = random_number_array(1e6);
.
```

2 – Vectorized Version

The vectorized version of this operation is significantly simpler:

```
initial_values = random_number_array(1e6);
output_values = initial_values .* 2;         % vectorized multiplication
```

Note: The array **`output_values`** no longer needs to be initialized before the operation.

It is initialized at the same time that the multiplication step is executed on the array **`initial_values`**.

2 – Adding a Timer

The last piece in our test is to wrap a timer around the code.

In MATLAB, the `tic` command starts a timer and `toc` ends it.

```
timer_start = tic;
.
.
```

2 – Adding a Timer

The last piece in our test is to wrap a timer around the code.

In MATLAB, the `tic` command starts a timer and `toc` ends it.

```
timer_start = tic;
<some code>
.
```

2 – Adding a Timer

The last piece in our test is to wrap a timer around the code.

In MATLAB, the `tic` command starts a timer and `toc` ends it.

```
timer_start = tic;
<some code>
total_time = toc(timer_start)
```

We now have all the pieces to put together an actual experiment!

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
.
.
.
.
.
.
.
```

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) ); % initialize output array
.
.
.
.
.
.
```

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
% --------------- TIME THE OPERATION --------------- %
matlab_for_loop_timer = tic;                        % start a timer
.
.
.
.
```

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
% --------------- TIME THE OPERATION --------------- %
matlab_for_loop_timer = tic;
for i = 1 : length(initial_values)                  % outline the loop
  .
end
.
```

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
% --------------- TIME THE OPERATION --------------- %
matlab_for_loop_timer = tic;
for i = 1 : length(initial_values)
  output_values(i) = initial_values(i) * 2;     % loop multiplication
end
.
```

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
% --------------- TIME THE OPERATION --------------- %
matlab_for_loop_timer = tic;
for i = 1 : length(initial_values)
  output_values(i) = initial_values(i) * 2;
end
matlab_for_loop_time = toc(matlab_for_loop_timer)    % end timer
```

The above code evaluates how long it takes to multiply 1 million elements by 2 ***using a for loop***.

2 – Testing the Loop

```
initial_values = random_number_array(1e6);
output_values = zeros( size(initial_values) );
% --------------- TIME THE OPERATION --------------- %
matlab_for_loop_timer = tic;
for i=1 : length(initial_values)
  output_values(i) = initial_values(i) * 2;
end
matlab_for_loop_time = toc(matlab_for_loop_timer)
```

```
matlab_for_loop_time =
     1.5195e-02  % [s]
```

Command Window Output

2 – Testing Vectorization

```
initial_values = random_number_array(1e6);
.
.
.
.
```

2 – Testing Vectorization

```
initial_values = random_number_array(1e6);
% --------------- TIME THE OPERATION --------------- %
vectorization_timer = tic;
.
.
```

2 – Testing Vectorization

✓ L.O.1
➩ L.O.2
❏ L.O.3
❏ L.O.4

```
initial_values = random_number_array(1e6);
% --------------- TIME THE OPERATION ---------------- %
vectorization_timer = tic;
output_values = initial_values .* 2;         % vectorized multiplication
.
```

2 – Testing Vectorization

```
initial_values = random_number_array(1e6);
% --------------- TIME THE OPERATION ---------------- %
vectorization_timer = tic;
output_values = initial_values .* 2;         % vectorized multiplication
vectorization_time = toc(vectorization_timer)
```

The above code evaluates how long it takes to multiply 1 million elements by 2 ***using vectorized code***.

2 – Testing Vectorization

```
initial_values = random_number_array(1e6);
% --------------- TIME THE OPERATION ---------------- %
vectorization_timer = tic;
output_values = initial_values .* 2;         % vectorized multiplication
vectorization_time = toc(vectorization_timer)
```

```
vectorization_time =
     5.7600e-04  % [s]
```

Command Window Output

2 – Comparing Results

```
matlab_for_loop_time =
     1.5195e-02  % [s]
.
.
```

Command Window Output

2 – Comparing Results

```
matlab_for_loop_time =
     1.5195e-02  % [s]
vectorization_time =
     5.7600e-04  % [s]
```

Command Window Output

In this example, vectorization is $\approx$ 26x faster!

```
>> matlab_for_loop_time / vectorization_time =
     2.6380e+01
```

Command Window

3 – Vectorized Array Operations

MATLAB has specific syntax for writing vectorized operations.

Rules

The + and – operators don't require any modification

The `*`, `/`, and `^` operators require the addition of a preceding period to indicate that it should be vectorized. For example:

```
a = u + v;   % [ u(1) + v(1), etc. ]
.
```

```
.
.
.
```

3 – Vectorized Array Operations

MATLAB has specific syntax for writing vectorized operations.

Rules

The + and – operators don't require any modification

The *, /, and ^ operators require the addition of a preceding period to indicate that it should be vectorized. For example:

```
a = u + v;
b = u - v;   % [ u(1) - v(1), etc. ]
```

```
.
.
.
```

3 – Vectorized Array Operations

MATLAB has specific syntax for writing vectorized operations.

Rules

The + and – operators don't require any modification

The *, /, and ^ operators require the addition of a preceding period to indicate that it should be vectorized. For example:

```
a = u + v;
b = u - v;
```

```
c = u .* v; % [ u(1) * v(1), etc. ]
.
.
```

3 – Vectorized Array Operations

MATLAB has specific syntax for writing vectorized operations.

Rules

The + and – operators don't require any modification

The *, /, and ^ operators require the addition of a preceding period to indicate that it should be vectorized. For example:

```
a = u + v;
b = u - v;
```

```
c = u .* v;
d = u ./ v; % [ u(1) / v(1), etc. ]
.
```

3 – Vectorized Array Operations

MATLAB has specific syntax for writing vectorized operations.

Rules

The + and – operators don't require any modification

The *, /, and ^ operators require the addition of a preceding period to indicate that it should be vectorized. For example:

```
a = u + v;
b = u - v;
```

```
c = u .* v;
d = u ./ v;
e = u .^ v; % [ u(1) ^ v(1), etc. ]
```

4 – Applying this Knowledge

Knowing the basics of writing vectorized operations, we can build upon the **`dot_product`** function from Lecture 2.1.

The old function was hardcoded to multiply all three components and then add them together:

```
dot_product = @(v1, v2) (v1(1)*v2(1)) + (v1(2)*v2(2)) + (v1(3)*v2(3));
```

Element-wise multiplication is easily replaced with the vectorized operation between the arrays **`v1`** and **`v2`**.

4 – First Improvement

This returns another vector containing the results of each element-wise multiplication:

```
v = v1 .* v2;
```

4 – First Improvement

This returns another vector containing the results of each element-wise multiplication:

```
v = v1 .* v2;       % v = [ v1(1) * v2(1), v1(2) * v2(2), v1(3) * v2(3) ]
```

To return the dot product, the **sum** command can be applied to add each component together:

```
s = sum(v1 .* v2);    % s = v1(1) * v2(1) + v1(2) * v2(2) + v1(3) * v2(3)
```

4 – Mx3 Inputs

Previously, the inputs were both `1x3` vectors.

This function can be expanded to accommodate solving for `M` dot products by passing `v1` and `v2` as `Mx3` arrays.

For example, consider the following two inputs:

```
v1 = [1, 2, 3; ...  % 2x3 array (each row is a 1x3 vector)
      4, 5, 6];
.
.
```

4 – **Mx3** Inputs

Previously, the inputs were both **1x3** vectors.

This function can be expanded to accommodate solving for **M** dot products by passing **v1** and **v2** as **Mx3** arrays.

For example, consider the following two inputs:

```
v1 = [1, 2, 3; ...
      4, 5, 6];
v2 = [6, 5, 4; ...  % 2x3 array (each row is a 1x3 vector)
      3, 2, 1];
```

4 – Extracting Component Columns

Then, the **x**-, **y**-, and **z**-components of **v1** and **v2** are all **Mx1** vectors. In our example:

```
v1(:,1) = [1; ...    % x-components of both 1x3 vectors in v1
           4]
.
.
.
.
```

4 – Extracting Component Columns

✓ L.O.1
✓ L.O.2
✓ L.O.3
⇨ L.O.4

Then, the **x**-, **y**-, and **z**-components of **v1** and **v2** are all **Mx1** vectors. In our example:

```
v1(:,1) = [1; ...
           4]
v1(:,2) = [2; ...    % y-components of both 1x3 vectors in v1
           5]
.
.
```

4 – Extracting Component Columns

Then, the **`x`**-, **`y`**-, and **`z`**-components of **`v1`** and **`v2`** are all **`Mx1`** vectors. In our example:

```
v1(:,1) = [1; ...
           4]
v1(:,2) = [2; ...
           5]
v1(:,3) = [3; ...    % z-components of both 1x3 vectors in v1
           6]
```

Note: the colon in parentheses indicates accessing the entirety of **`v1`** along that dimension — every ***row*** in this example.

4 – A Preliminary Rewrite

✓ L.O.1
✓ L.O.2
✓ L.O.3
⇨ L.O.4

At this point, the component-wise multiplication in the dot product can be vectorized as follows:

```
dot_product = @(v1, v2) (v1(:,1) .* v2(:,1)) ... % multiply x-components
                     .
                     .
```

4 – A Preliminary Rewrite

At this point, the component-wise multiplication in the dot product can be vectorized as follows:

```
dot_product = @(v1, v2) (v1(:,1) .* v2(:,1)) ...
                      + (v1(:,2) .* v2(:,2)) ... % multiply y-components
                      .
```

4 – A Preliminary Rewrite

At this point, the component-wise multiplication in the dot product can be vectorized as follows:

```
dot_product = @(v1, v2) (v1(:,1) .* v2(:,1)) ...
                      + (v1(:,2) .* v2(:,2)) ...
                      + (v1(:,3) .* v2(:,3));    % multiply z-components
```

This mirrors the original form for the **`dot_product`** function.

This can be further improved using vectorized multiplication.

4 – Using the `sum` Command

`v1` and **`v2`** are both **`Mx3`**, so the operation **`v1.*v2`** returns an **`Mx3`** array from each element-wise multiplication.

```
v1 .* v2 = [v1(:,1) .* v2(:,1), ...
                .
                .
```

4 – Using the **`sum`** Command

`v1` and **`v2`** are both **`Mx3`**, so the operation **`v1.*v2`** returns an **`Mx3`** array from each element-wise multiplication.

```
v1 .* v2 = [v1(:,1) .* v2(:,1), ...
            v1(:,2) .* v2(:,2), ...
            .
```

4 – Using the `sum` Command

`v1` and **`v2`** are both **`Mx3`**, so the operation **`v1.*v2`** returns an **`Mx3`** array from each element-wise multiplication.

```
v1 .* v2 = [v1(:,1) .* v2(:,1), ...
            v1(:,2) .* v2(:,2), ...
            v1(:,3) .* v2(:,3)];
```

The **`sum`** command can be used to condense this into an **`Mx1`** vector of every dot product: **`sum`** along each row.

```
sum(a, <dimension>) % sums a along the specified dimension
```

4 – Specifying an Axis for `sum`

The `sum` documentation details the options for specifying the dimension along which to add elements:

- `sum(a,1)` — sum along each column
- `sum(a,2)` — sum along each row
- `sum(a,'all')` — sum every element

Therefore, the dimension `2` is appropriate for our application.

4 – Full Vectorized Rewrite

All together, the new and improved **`dot_product`** anonymous function is written as follows:

```
dot_product = @(v1, v2)
```

4 – Full Vectorized Rewrite

All together, the new and improved **`dot_product`** anonymous function is written as follows:

```
dot_product = @(v1, v2) sum(v1 .* v2
```

4 – Full Vectorized Rewrite

All together, the new and improved **dot_product** anonymous function is written as follows:

```
dot_product = @(v1, v2) sum(v1 .* v2, 2);
```

This function accepts two **Mx3** arrays as inputs.

It computes **M** dot products between each **1x3** vector.

And most importantly, it is very easy to read!

4 – Bigger Random Inputs

Consider the following update to the earlier random array function:

```
random_number_array = @(size) rand(size);
```

Then, we can define two input arrays, as follows:

```
v1 = random_number_array([100, 3]);
.
```

4 – Bigger Random Inputs

Consider the following update to the earlier random array function:

```
random_number_array = @(size) rand(size);
```

Then, we can define two input arrays, as follows:

```
v1 = random_number_array([100, 3]);
v2 = random_number_array([100, 3]);
```

Both **`v1`** and **`v2`** are now **`Mx3`** arrays.

4 – New and Improved Demo

These `Mx3` arrays can be tested in the fully vectorized form of the `dotProduct` function:

```
>> v1 = random_number_array([100, 3]);
.
.
```

Command Window

4 – New and Improved Demo

These **Mx3** arrays can be tested in the fully vectorized form of the **dotProduct** function:

```
>> v1 = random_number_array([100, 3]);
>> v2 = random_number_array([100, 3]);
.
```

Command Window

4 – New and Improved Demo

These **`Mx3`** arrays can be tested in the fully vectorized form of the **`dotProduct`** function:

```
>> v1 = random_number_array([100, 3]);
>> v2 = random_number_array([100, 3]);
>> dot_products =
```

Command Window

4 – New and Improved Demo

These **`Mx3`** arrays can be tested in the fully vectorized form of the **`dotProduct`** function:

```
>> v1 = random_number_array([100, 3]);
>> v2 = random_number_array([100, 3]);
>> dot_products = dot_product(v1, v2)
```

Command Window

4 – New and Improved Demo

These `Mx3` arrays can be tested in the fully vectorized form of the `dotProduct` function:

```
>> v1 = random_number_array([100, 3]);
>> v2 = random_number_array([100, 3]);
>> dot_products = dot_product(v1, v2)
<100x1 vector output>
```

Command Window

5 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4

This lecture covered:

✓ What vectorization is

The ability for code to execute a single operation on a whole set of data.

✓ Why vectorization could be useful in engineering

It enables large problems to be solved more quickly and in a much more visually clean manner, which simplifies the code for comprehension and debugging.

5 – Summary

✓ Vectorized array operations in MATLAB

For addition and subtraction, no modification is required. Multiplication, division, and exponentiation require the addition of a preceding period to indicate a vectorized array operation.

✓ Vectorization of the dot product

In the previous lecture, we wrote a custom function to take the dot product between two vectors. Now, we use the directional **`sum`** command and element-wise vectorized multiplication to solve the dot product for *many* pairs of vectors.